序

在這本書問世之前，我就已經看過霍春陽的 Vue.js 3 原始碼解讀，當時就很欣賞他對技術細節的專注和投入。後來春陽為 Vue.js 3 上傳了大量修正，修復了一些非常深層的渲染更新 bug，為 Vue.js 3 做出了很多貢獻，成為了官方團隊成員。春陽對 Vue.js 3 原始碼的理解來自參與原始碼的維護中，這是深入理解開源專案最有難度但也最有效的途徑。也因此這本書對 Vue.js 3 技術細節的分析非常可靠，對於需要深入理解 Vue.js 3 的使用者會有很大的幫助。

春陽對 Vue.js 的高層設計思維的理解也非常精準，並且在框架的設計權衡層面有自己的深入思考。這可能是這本書最不同於市面上其他純粹的「原始碼分析」類型圖書的地方：它從高層的設計角度探討框架需要關注的問題，從而幫助讀者更好地理解一些具體的實作為何要做出這樣的選擇。

前端是一個變化很快的領域，新的技術不斷出現，Vue.js 本身也在不斷地進化，我們還會繼續探索最佳化的實作細節。但即使拋開具體的實作，這本書也可以作為現代前端框架設計的一個非常有價值的參考。

尤雨溪，Vue.js 作者

前言

Vue.js 作為最流行的前端框架之一，在 2020 年 9 月 18 日，正式迎來了它的 3.0 版本。得益於 Vue.js 2 的設計經驗，Vue.js 3.0 不僅帶來了諸多新特性，還在框架設計與實作上做了很多創新。在一定程度上，我們可以認為 Vue.js 3.0「還清」了在 Vue.js 2 中欠下的技術債務。

在我看來，Vue.js 3.0 是一個非常成功的專案。它秉承了 Vue.js 2 的易用性。同時，相比 Vue.js 2，Vue.js 3.0 甚至做到了使用更少的程式碼來實作更多的功能。

Vue.js 3.0 在模組的拆分和設計上做得非常合理。模組之間的耦合度非常低，很多模組可以獨立安裝使用，而不需要依賴完整的 Vue.js 執行，例如 `@vue/reactivity` 模組。

Vue.js 3.0 在設計內建組件和模組時也花費了很多精力，配合建構工具以及 Tree-Shaking 機制，實作了內建能力的按照需要時引入，從而實作了使用者 bundle 的體積最小化。

Vue.js 3.0 的擴展能力非常強，我們可以撰寫自訂的渲染器，甚至可以撰寫編譯器外掛程式來自訂模板語法。同時，Vue.js 3.0 在使用者體驗上也下足了功夫。

Vue.js 3.0 的優點包括但不限於上述這些內容。既然 Vue.js 3.0 的優點如此之多，那麼框架設計者是如何設計並實作這一切的呢？實際上，理解 Vue.js 3.0 的核心設計思維非常重要。它不僅能夠讓我們更加從容地面對複雜問題，還能夠指導我們在其他領域進行架構設計。

另外，Vue.js 3.0 中很多功能的設計需要謹遵規範。例如，想要使用 `Proxy` 實作完善的響應系統，就必須從 ECMAScript 規範入手，而 Vue.js 的模板解析器則遵從 WHATWG 的相關規範。所以，在理解 Vue.js 3.0 核心設計思維的同時，我們還能夠間接掌握閱讀和理解規範，並據此撰寫程式碼。

讀者對象

本書的目標讀者包括：

- 對 Vue.js 2/3 具有上手經驗，且希望進一步理解 Vue.js 框架設計原理的開發人員。
- 沒有使用過 Vue.js，但對 Vue.js 框架設計感興趣的前端開發人員。

本書內容

本書內容並非「原始碼解讀」，而是建立在筆者對 Vue.js 框架設計的理解之上，以由簡入繁的方式介紹如何實作 Vue.js 中的各個功能模組。

本書將盡可能地從規範出發，實作功能完善且嚴謹的 Vue.js 功能模組。例如，透過閱讀 ECMAScript 規範，基於 `Proxy` 實作一個完善的響應系統；透過閱讀 WHATWG 規範，實作一個類似 HTML 語法的模板解析器，並在此基礎上實作一個支援外掛程式架構的模板編譯器。

除此之外，本書還會討論以下內容：

- 框架設計的核心要素以及框架設計過程中要做出的權衡；
- 三種常見的虛擬 DOM（Virtual DOM）的 Diff 演算法；
- 模組化的實作與 Vue.js 內建模組的原理；
- 伺服端渲染、使用者端渲染、同構渲染之間的差異，以及同構渲染的原理。

本書結構

本書分為 6 篇，共 18 章，各章的簡介如下。

- 第一篇（框架設計概覽）：共 3 章。
  - 第 1 章主要討論了命令式和聲明式這兩種的差異，以及二者對框架設計的影響，還討論了虛擬 DOM 的效能狀況，最後介紹了執行時和編譯時的相關知識，並介紹了 Vue.js 3.0 是一個執行時 + 編譯時的框架。
  - 第 2 章主要從使用者的開發體驗、控制框架程式碼的體積、Tree-Shaking 的運作機制、框架產物、屬性開關、錯誤處理、TypeScript 支援等幾個方面出發，討論了框架設計者在設計框架時應該考慮的內容。
  - 第 3 章從全域視角介紹 Vue.js 3.0 的設計思路，以及各個模組之間是如何協作的。

- 第二篇（響應系統）：共 3 章。
  - 第 4 章從宏觀視角講述了 Vue.js 3.0 中響應系統的實作機制。從副作用函數開始，逐步實作一個完善的響應系統，還講述了運算屬性和 `watch` 的實作原理，同時討論了在實作響應系統的過程中所遇到的問題，以及相應的解決方案。
  - 第 5 章從 ECMAScript 規範入手，從最基本的 `Proxy`、`Reflect` 以及 JavaScript 物件的運作原理開始，逐步討論了使用 `Proxy` 代理 JavaScript 物件的方式。
  - 第 6 章主要討論了 `ref` 的概念，並基於 `ref` 實作原始值的響應式方案，還討論了如何使用 `ref` 解決響應遺失問題。
- 第三篇（渲染器）：共 5 章。
  - 第 7 章主要討論了渲染器與響應系統的關係，講述了兩者如何配合工作完成頁面更新，還討論了渲染器中的一些基本名詞和概念，以及自訂渲染器的實作與應用。
  - 第 8 章主要討論了渲染器載入與更新的實作原理，其中包括子節點的處理、屬性的處理和事件的處理。當載入或更新組件類型的虛擬節點時，還要考慮組件生命週期函數的處理等。
  - 第 9 章主要討論了「簡單 Diff 演算法」的運作原理。
  - 第 10 章主要討論了「雙端 Diff 演算法」的運作原理。
  - 第 11 章主要討論了「快速 Diff 演算法」的運作原理。
- 第四篇（組件化）：共 3 章。
  - 第 12 章主要討論了組件的實作原理，介紹了組件自身狀態的初始化，以及由自身狀態變化引起的組件自更新，還介紹了組件的外部狀態（`props`）、由外部狀態變化引起的被動更新，以及組件事件和插槽的實作原理。
  - 第 13 章主要介紹了非同步組件和函數式組件的運作機制和實作原理。對於非同步組件，我們還討論了超時與錯誤處理、延遲展示 Loading 組件、載入重試等內容。
  - 第 14 章主要介紹了 Vue.js 內建的三個組件的實作原理，即 KeepAlive、Teleport 和 Transition 組件。

- 第五篇（編譯器）：共 3 章。
  - 第 15 章首先討論了 Vue.js 模板編譯器的運作流程，接著討論了 `parser` 的實作原理與狀態機，以及 AST 的轉換與外掛程式化架構，最後討論了生成渲染函數程式碼的具體實作。
  - 第 16 章主要討論了如何實作一個符合 WHATWG 組織的 HTML 解析規範的解析器，內容涵蓋解析器的文本模式、文本模式對解析器的影響，以及如何使用遞迴下降演算法建構模板 AST。在解析文本內容時，我們還討論了如何根據規範解碼字元引用。
  - 第 17 章主要討論了 Vue.js 3.0 中模板編譯最佳化的相關內容。具體包括：`Block` 樹的更新機制、動態節點的收集、靜態提升、預字串化、暫存內嵌事件處理函數、`v-once` 等最佳化機制。
- 第六篇（伺服端渲染）：1 章。
  - 第 18 章主要討論了 Vue.js 同構渲染的原理。首先探討了 CSR、SSR 以及同構渲染等方案各自的優缺點，然後探討了 Vue.js 進行伺服端渲染和使用者端啟動的原理，最後總結了撰寫同構程式碼時的注意事項。

原始程式碼及勘誤

在學習本書時，書中所有程式碼均可透過手動輸入程式碼進行試驗和學習，你也可以從 GitHub（HcySunYang）下載所有原始程式碼[1]。我將盡最大努力確保正文和來源程式碼無誤。但金無足赤，書中難免存在一些錯誤。如果讀者發現任何錯誤，包括但不限於別字、程式碼片段、描述有誤等，請及時回饋給我。本書勘誤請至 GitHub（HcySunYang）查看或上傳[2]。

1 或存取圖靈社區本書主頁下載。——編者注
2 也可存取圖靈社區本書主頁查看或上傳勘誤。——編者注

第 1 章 權衡的藝術

「框架設計裡到處都展現了權衡的藝術。」

在深入討論 Vue.js 3 各個模組的實作思路和細節之前，我認為有必要先來討論視圖層框架設計方面的內容。為什麼呢？這是因為當我們設計一個框架的時候，框架本身的各個模組之間並不是相互獨立的，而是相互關聯、相互制約的。因此作為框架設計者，一定要對框架的定位和方向擁有全域的掌握，這樣才能做好後續的模組設計和拆分。同樣，作為學習者，我們在學習框架的時候，也應該從全域的角度對框架的設計擁有清晰的認知，否則很容易被細節困住，看不清全貌。

另外，從範例的角度來看，我們的框架應該設計成命令式的還是聲明式的呢？這兩種範例有何優缺點？我們能否汲取兩者的優點？除此之外，我們的框架要設計成純執行期的還是純編譯期的，甚至是執行期＋編譯期的呢？它們之間又有何差異？優缺點分別是什麼？這裡面都展現了「權衡」的藝術。

1.1 命令式和聲明式

從範例上來看，視圖層框架通常分為命令式和聲明式，它們各有優缺點。作為框架設計者，應該對兩種範例都有足夠的認知，這樣才能做出正確的選擇，甚至想辦法汲取兩者的優點並將其整合。

接下來，我們先來看看命令式框架和聲明式框架的概念。早年間流行的 jQuery 就是典型的命令式框架。命令式框架的一大特點就是**著重過程**。例如，我們把下面這段話翻譯成對應的程式碼：

```
- 獲取 id 為 app 的 div 標籤
- 它的文本內容為 hello world
- 為其綁定點擊事件
- 當點擊時彈出提示：ok
```

對應的程式碼為：

```
$('#app') // 獲取 div
  .text('hello world') // 設定文本內容
  .on('click', () =>{ alert('ok') }) // 綁定點擊事件
```

以上就是 jQuery 的程式碼範例，考慮到有些讀者可能沒有用過 jQuery，因此我們再用原生 JavaScript 來實作同樣的功能：

```
const div = document.querySelector('#app') // 獲取 div
div.innerText = 'hello world' // 設定文本內容
div.addEventListener('click', ()=>{alert('ok')}) // 綁定點擊事件
```

可以看到，自然語言描述能夠與程式碼產生一一對應的關係，程式碼本身描述的是「做事的過程」，這符合我們的邏輯直覺。

那麼，什麼是聲明式框架呢？與命令式框架更加著重過程不同，聲明式框架更加著重結果。結合 Vue.js，我們來看看如何實作上面自然語言描述的功能：

```
<div @click="() => alert('ok')">hello world</div>
```

這段類似 HTML 的模板就是 Vue.js 實作如上功能的方式。可以看到，我們提供的是一個「結果」，至於如何實作這個「結果」，我們並不關心，這就像我們在告訴 Vue.js：「嘿，Vue.js，看到沒，我要的就是一個 `div`，文本內容是 `hello world`，它有綁定一個事件，你幫我搞定吧。」至於實作該「結果」的**過程**，則是由 Vue.js 幫我們完成的。換句話說，Vue.js 幫我們封裝了**過程**。因此，我們能夠猜到 Vue.js 的內部實作一定是**命令式**的，而暴露給使用者的卻是**聲明式**。

1.2 效能與可維護性的權衡

命令式和聲明式各有優缺點，在框架設計方面，則展現在效能與可維護性之間的權衡。這裡我們先拋出一個結論：**聲明式程式碼的效能不優於命令式程式碼的效能**。

還是拿上面的例子來說，假設現在我們要將 `div` 標籤的文本內容修改為 `hello vue3`，那麼如何用命令式程式碼實作呢？很簡單，因為我們明確知道要修改的是什麼，所以直接呼叫相關命令操作即可：

```
div.textContent = 'hello vue3' // 直接修改
```

現在思考一下，還有沒有其他辦法比上面這句程式碼的效能更好？答案是「沒有」。可以看到，理論上命令式程式碼可以做到極致的效能最佳化，因為我們明確知道哪些發生了變更，只做必要的修改就行了。但是聲明式程式碼不一定能做到這一點，因為它描述的是結果：

```
<!-- 之前： -->
<div @click="() => alert('ok')">hello world</div>
<!-- 之後： -->
<div @click="() => alert('ok')">hello vue3</div>
```

對於框架來說，為了實作最優的更新效能，它需要找到前後的差異並只更新變化的地方，但是最終完成這次更新的程式碼仍然是：

```
1  div.textContent = 'hello vue3' // 直接修改
```

如果我們把直接修改的效能消耗定義為 A，把找出差異的效能消耗定義為 B，那麼有：

- 命令式程式碼的更新效能消耗＝ A
- 聲明式程式碼的更新效能消耗＝ B ＋ A

可以看到，聲明式程式碼會比命令式程式碼多出找出差異的效能消耗，因此最理想的情況是，當找出差異的效能消耗為 0 時，聲明式程式碼與命令式程式碼的效能相同，但是無法做到超越，**畢竟框架本身就是封裝了命令式程式碼才實作了面向使用者的聲明式**。這符合前文中提供的效能結論：聲明式程式碼的效能不優於命令式程式碼的效能。

既然在效能層面命令式程式碼是更好的選擇，那麼為什麼 Vue.js 要選擇聲明式的設計方案呢？原因就在於聲明式程式碼的可維護性更強。從上面例子的程式碼中我們也可以感受到，在採用命令式程式碼開發的時候，我們需要維護實作目標的整個**過程**，包括要手動完成 DOM 元素的建立、更新、刪除等操作。而聲明式程式碼展示的就是我們要的**結果**，看上去更加直觀，至於做事的過程，並不需要我們關心，Vue.js 都為我們封裝好了。

這就展現了我們在框架設計上要做出的關於可維護性與效能之間的權衡。在採用聲明式提升可維護性的同時，效能就會有一定的損失，而框架設計者要做的就是：**在保持可維護性的同時讓效能損失最小化**。

1.3 虛擬 DOM 的效能到底如何

考慮到有些讀者可能不知道什麼是虛擬 DOM，這裡我們不會對其做深入討論，但這既不影響你理解本節內容，也不影響你閱讀後續章節。如果實在看不明白，也沒關係，至少有個印象，等後面我們深入講解虛擬 DOM 後再回來看這裡的內容，相信你會有不同的感受。

前文說到，**聲明式程式碼的更新效能消耗＝找出差異的效能消耗＋直接修改的效能消耗**，因此，如果我們能夠最小化**找出差異的效能消耗**，就可以讓聲明式程式碼的效能無限接近命令式程式碼的效能。而所謂的虛擬 DOM，就是為了**最小化**找出差異這一步的效能消耗而出現的。

至此，相信你也應該清楚一件事了，那就是採用虛擬 DOM 的更新技術的效能**理論上**不可能比原生 JavaScript 操作 DOM 更高。這裡我們強調了**理論上**三個字，因為這很關鍵，為什麼呢？因為在大部分情況下，**我們很難寫出絕對最佳化的命令式程式碼**，尤其是當應用程式的規模很大的時候，即使你寫出了極致最佳化的程式碼，也一定耗費了巨大的精力，這時的投入產出比其實並不高。

那麼，有沒有什麼辦法能夠讓我們不用付出太多的努力（寫聲明式程式碼），還能夠保證應用程式的效能下限，讓應用程式的效能不至於太差，甚至想辦法逼近命令式程式碼的效能呢？這其實就是虛擬 DOM 要解決的問題。

不過前文中所說的原生 JavaScript 實際上指的是像 `document.createElement` 之類的 DOM 操作方法，並不包含 `innerHTML`，因為它比較特殊，需要單獨討論。在早年使用 jQuery 或者直接使用 JavaScript 撰寫頁面的時候，使用 `innerHTML` 來操作頁面非常常見。其實我們可以思考一下：使用 `innerHTML` 操作頁面和虛擬 DOM 相比效能如何？ `innerHTML` 和 `document.createElement` 等 DOM 操作方法有何差異？

先來看第一個問題，為了比較 `innerHTML` 和虛擬 DOM 的效能，我們需要瞭解它們建立、更新頁面的過程。對於 `innerHTML` 來說，為了建立頁面，我們需要撰寫一段 HTML 字串：

```
const html = `
<div><span>...</span></div>
`
```

接著將該字串賦值給 DOM 元素的 `innerHTML` 屬性：

```
div.innerHTML = html
```

然而這句話遠沒有看上去那麼簡單。為了渲染出頁面，首先要把字串解析成 DOM 樹，這是一個 DOM 層面的計算。我們知道，涉及 DOM 的運算要遠比 JavaScript 層面的計算效能差，這有一個效能運算結果可供參考，如圖 1-1 所示。

▲ 圖 1-1 效能運算結果

在圖 1-1 中，上邊是純 JavaScript 層面的計算，迴圈 10000 次，每次建立一個 JavaScript 物件並將其新增到程式中；下邊是 DOM 操作，每次建立一個 DOM 元素並將其新增到頁面中。效能運算結果輸出，純 JavaScript 層面的操作要比 DOM 操作快得多，它們不在同個量級上。基於這個背景，我們可以用一個公式來表達透過 innerHTML 建立頁面的效能：**HTML 字串拼接的計算量＋ innerHTML 的 DOM 計算量**。

接下來，我們討論虛擬 DOM 在建立頁面時的效能。虛擬 DOM 建立頁面的過程分為兩步：第一步是建立 JavaScript 物件，這個物件可以理解為真實 DOM 的描述；第二步是遍歷執行虛擬 DOM 樹並建立真實 DOM。我們同樣可以用一個公式來表達：**建立 JavaScript 物件的計算量＋建立真實 DOM 的計算量**。

圖 1-2 直觀地對比了 innerHTML 和虛擬 DOM 在建立頁面時的效能。

	虛擬 DOM	innerHTML
純 JavaScript 運算	• 創立 JavaScript 物件 (VNode)	• 渲染 HTML 程式碼
DOM 運算	• 新建所有 DOM 元素	• 新建所有 DOM 元素

▲ 圖 1-2　innerHTML 和虛擬 DOM 在建立頁面時的效能

可以看到，無論是純 JavaScript 層面的計算，還是 DOM 層面的計算，其實兩者差距不大。這裡我們從宏觀的角度只看量級上的差異。如果在同一個量級，則認為沒有差異。在建立頁面的時候，都需要新建所有 DOM 元素。

剛剛我們討論了建立頁面時的效能情況，大家可能會覺得虛擬 DOM 相比 innerHTML 沒有優勢可言，甚至細究的話效能可能會更差。別著急，接下來我們看看它們在更新頁面時的效能。

使用 innerHTML 更新頁面的過程是**重新建構 HTML 字串，再重新設定 DOM 元素的 innerHTML 屬性**，這其實是在說，哪怕我們只更改了一個文字，也要重新設定 innerHTML 屬性。而重新設定 innerHTML 屬性就等於**銷毀所有舊的 DOM 元素，再全部建立新的 DOM 元素**。再來看虛擬 DOM 是如何更新頁面的。它需要重新建立 JavaScript 物件（虛擬 DOM 樹），然後比較新舊虛擬 DOM，找到變化的元素並更新它。圖 1-3 可作為對照。

	虛擬 DOM	innerHTML
純 JavaScript 運算	• 建立新的 JavaScript 物件 + Diff	• 渲染 HTML 程式碼
DOM 運算	• 必要的 DOM 更新	• 銷毀所有舊 DOM • 新建所有新 DOM

▲ 圖 1-3 虛擬 DOM 和 innerHTML 在更新頁面時的效能

可以發現，在更新頁面時，虛擬 DOM 在 JavaScript 層面的運算要比建立頁面時多出一個 Diff 的效能消耗，然而它畢竟也是 JavaScript 層面的運算，所以不會產生數量級的差異。再觀察 DOM 層面的運算，可以發現虛擬 DOM 在更新頁面時只會更新必要的元素，但 innerHTML 需要全部更新。這時虛擬 DOM 的優勢就展現出來了。

另外，我們發現，當更新頁面時，影響虛擬 DOM 的效能因素與影響 innerHTML 的效能因素不同。對於虛擬 DOM 來說，無論頁面多大，都只會更新變化的內容，而對於 innerHTML 來說，頁面越大，就意味著更新時的效能消耗越大。如果加上效能因素，那麼最終它們在更新頁面時的效能如圖 1-4 所示。

	虛擬 DOM	innerHTML
純 JavaScript 運算	• 建立新的 JavaScript 物件 + Diff	• 渲染 HTML 程式碼
DOM 運算	• 必要的 DOM 更新	• 銷毀所有舊 DOM • 新建所有新 DOM
效能因素	• 與資料變化量相關	• 與模板大小相關

▲ 圖 1-4 虛擬 DOM 和 innerHTML 在更新頁面時的效能（加上效能因素）

基於此，我們可以粗略地總結一下 innerHTML、虛擬 DOM 以及原生 JavaScript（指 createElement 等方法）在更新頁面時的效能，如圖 1-5 所示。

▲ 圖 1-5　innerHTML、虛擬 DOM 以及原生 JavaScript 在更新頁面時的效能

我們分了幾個維度：心智負擔、可維護性和效能。其中原生 DOM 操作方法的心智負擔最大，因為你要手動建立、刪除、修改大量的 DOM 元素。但它的效能是最高的，不過為了使其效能最佳，我們同樣要承受巨大的心智負擔。另外，以這種方式撰寫的程式碼，可維護性也極差。而對於 innerHTML 來說，由於我們撰寫頁面的過程有一部分是透過拼接 HTML 標籤來實作的，這有點接近聲明式的意思，但是拼接字串也是有一定心智負擔的，而且對於事件綁定之類的事情，我們還是要使用原生 JavaScript 來處理。如果 innerHTML 模板很大，則其更新頁面的效能最差，尤其是在只有少量更新時。最後，我們來看看虛擬 DOM，它是聲明式的，因此心智負擔小，可維護性強，效能雖然比不上極致最佳化的原生 JavaScript，但是在保證心智負擔和可維護性的前提下相當不錯。

至此，我們有必要思考一下：有沒有辦法做到，既聲明式地描述 UI，又具備原生 JavaScript 的效能呢？看上去有點魚與熊掌兼得的意思，我們會在下一章中繼續討論。

1.4 執行時和編譯時

當設計一個框架的時候，我們有三種選擇：純執行時的、執行時 + 編譯時的或純編譯時的。這需要你根據目標框架的特徵，以及對框架的期望，做出合適的決策。另外，為了做出合適的決策，你需要清楚地知道什麼是執行時，什麼是編譯時，它們各自有什麼特徵，它們對框架有哪些影響，本節將會逐步討論這些內容。

我們先聊聊純執行時的框架。假設我們設計了一個框架，它提供一個 Render 函數，使用者可以為該函數提供一個樹型結構的資料，然後 Render 函數會根據該物件遞迴地將資料渲染成 DOM 元素。我們規定樹型結構的資料如下：

```
const obj = {
  tag: 'div',
  children: [
    { tag: 'span', children: 'hello world' }
```

```
  ]
}
```

每個物件都有兩個屬性：tag 代表標籤名稱，children 既可以是一個陣列（代表子節點），也可以直接是一段文本（代表文本子節點）。接著，我們來實作 Render 函數：

```
function Render(obj, root) {
  const el = document.createElement(obj.tag)
  if (typeof obj.children === 'string') {
    const text = document.createTextNode(obj.children)
    el.appendChild(text)
  } else if (obj.children) {
    // 陣列，遞迴呼叫 Render，使用 el 作為 root 參數
    obj.children.forEach((child) => Render(child, el))
  }

  // 將元素新增到 root
  root.appendChild(el)
}
```

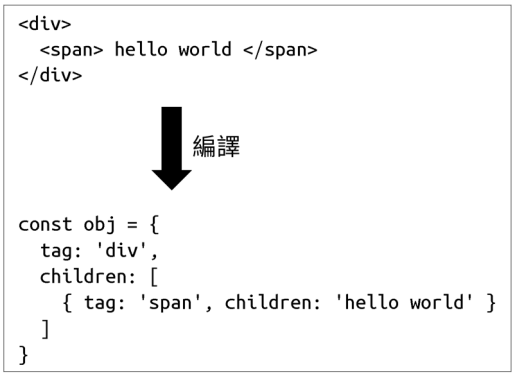
有了這個函數，使用者就可以這樣來使用它：

```
const obj = {
  tag: 'div',
  children: [
    { tag: 'span', children: 'hello world' }
  ]
}
// 渲染到 body 下
Render(obj, document.body)
```

在瀏覽器中執行上面這段程式碼，就可以看到我們預期的內容。

現在我們回過頭來思考一下使用者是如何使用 Render 函數的。可以發現，使用者在使用它渲染內容時，直接為 Render 函數提供了一個樹型結構的資料。這裡面不涉及任何額外的步驟，使用者也不需要學習額外的知識。但是有一天，你的使用者抱怨說：「手寫樹型結構的資料太麻煩了，而且不直觀，能不能支援用類似於 HTML 標籤的方式描述樹型結構的資料呢？」你看了看現在的 Render 函數，然後回答：「抱歉，暫不支援。」實際上，我們剛剛撰寫的框架就是一個**純執行時**的框架。

為了滿足使用者的需求，你開始思考，能不能引入編譯的手段，把 HTML 標籤編譯成樹型結構的資料，這樣不就可以繼續使用 Render 函數了嗎？思路如圖 1-6 所示。



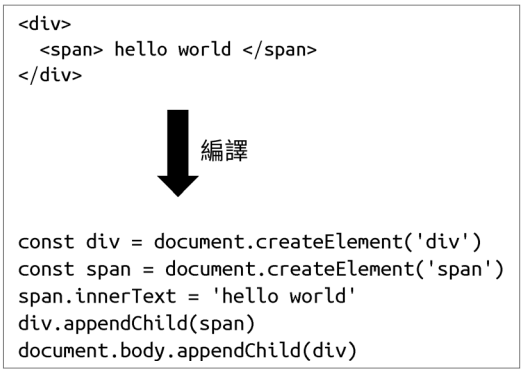
▲ 圖 1-6　把 HTML 標籤編譯成樹型結構的資料

為此，你撰寫了一個叫作 Compiler 的程式，它的作用就是把 HTML 標籤編譯成樹型結構的資料，於是交付給使用者去使用。那麼使用者該怎麼使用呢？其實這也是我們要思考的問題，最簡單的方式就是讓使用者分別呼叫 Compiler 函數和 Render 函數：

```
const html = `
<div>
  <span>hello world</span>
</div>
`
// 呼叫 Compiler 編譯得到樹型結構的資料
const obj = Compiler(html)
// 再呼叫 Render 進行渲染
Render(obj, document.body)
```

上面這段程式碼能夠很好地運作，這時我們的框架就變成了一個**執行時 + 編譯時**的框架。它既支援執行時，使用者可以直接提供資料從而無須編譯；又支援編譯時，使用者可以提供 HTML 標籤，我們將其編譯為資料後再交給執行時處理。準確地說，上面的程式碼其實是**執行時編譯**，意思是程式碼執行的時候才開始編譯，而這會產生一定的效能消耗，因此我們也可以在建構的時候就執行 Compiler 程式將使用者提供的內容編譯好，等到執行時就無須編譯了，這對效能是非常友好的。

不過，聰明的你一定意識到了另外一個問題：既然編譯器可以把 HTML 標籤編譯成資料，那麼能不能直接編譯成命令式程式碼呢？圖 1-7 展示了將 HTML 標籤編譯為命令式程式碼的過程。



▲ 圖 1-7 將 HTML 標籤編譯為命令式程式碼的過程

這樣我們只需要一個 Compiler 函數就可以了，連 Render 都不需要了。其實這就變成了一個**純編譯時**的框架，因為我們不支援任何執行時的內容，使用者的程式碼需透過編譯器編譯後才能執行。

我們用簡單的例子講解了框架設計層面的**執行時**、**編譯時**以及**執行時 + 編譯時**。我們發現，一個框架既可以是純執行時的，也可以是純編譯時的，還可以是既支援執行時又支援編譯時的。那麼，它們都有哪些優缺點呢？是不是既支援執行時又支援編譯時的框架最好呢？為了搞清楚這個問題，我們逐個分析。

首先是純執行時的框架。由於它沒有編譯的過程，因此我們沒辦法分析使用者提供的內容，但是如果加入編譯步驟，可能就大不一樣了，我們可以分析使用者提供的內容，看看哪些內容未來可能會改變，哪些內容永遠不會改變，這樣我們就可以在編譯的時候取得這些訊息，然後將其傳遞給 Render 函數，Render 函數得到這些訊息之後，就可以做進一步的最佳化了。然而，假如我們設計的框架是純編譯時的，那麼它也可以分析使用者提供的內容。由於不需要任何執行時，而是直接編譯成可執行的 JavaScript 程式碼，因此效能可能會更好，但是這種做法有損靈活性，即使用者提供的內容必須編譯後才能用。實際上，在這三個方向上業內都有探索，其中 Svelte 就是純編譯時的框架，但是它的真實效能可能達不到理論高度。Vue.js 3 仍然保持了執行時 + 編譯時的架構，在保持靈活性的基礎上能夠盡可能地最佳化。等到後面講解 Vue.js 3 的編譯最佳化相關內容時，你會看到 Vue.js 3 在保留執行時的情況下，其效能甚至不輸純編譯時的框架。

1.5 總結

在本章中，我們先討論了命令式和聲明式這兩種的差異，其中命令式更加關注過程，而聲明式更加關注結果。命令式在理論上可以做到極致最佳化，但是使用者要承受巨大的心智負擔；而聲明式能夠有效減輕使用者的心智負擔，但是效能上有一定的犧牲，框架設計者要想辦法盡量使效能損耗最小化。

接著，我們討論了虛擬 DOM 的效能，並提供了一個公式：**聲明式的更新效能消耗 = 找出差異的效能消耗 + 直接修改的效能消耗**。虛擬 DOM 的意義就在於使找出差異的效能消耗最小化。我們發現，用原生 JavaScript 操作 DOM 的方法（如 `document.createElement`）、虛擬 DOM 和 `innerHTML` 三者操作頁面的效能，不可以簡單地下定論，這與**頁面大小**、**變更部分的大小**都有關係，除此之外，與**建立頁面**還是**更新頁面**也有關係，選擇哪種更新策略，需要我們結合**心智負擔**、**可維護性**等因素綜合考慮。一番權衡之後，我們發現虛擬 DOM 是個還不錯的選擇。

最後，我們介紹了執行時和編譯時的相關知識，瞭解純執行時、純編譯時以及兩者都支援的框架各有什麼特點，並總結出 Vue.js 3 是一個編譯時 + 執行時的框架，它在保持靈活性的基礎上，還能夠透過編譯手段分析使用者提供的內容，從而進一步提升更新效能。